Estructurando código paralelo para clusters heterogéneos de CPUs/GPUs

Adrian Pousa[1] , Victoria Sanz[1,2], Armando De Giusti[1,2]

[1]III-LIDI, Facultad de Informática, UNLP, La Plata, Argentina
[2]CONICET, Ministerio de Ciencia, Tecnología e Innovación Productiva, Argentina
{apousa,vsanz,degiusti}@lidi.info.unlp.edu.ar

Resumen Los clusters de CPUs/GPUs se han vuelto habituales en HPC. Para aprovechar al máximo su potencia de cómputo, las aplicaciones deben desarrollarse combinando distintas herramientas de programación paralela, por esto el código se torna complejo y difícil de estructurar. En este trabajo describimos un esquema para estructurar código paralelo a ser ejecutado sobre un cluster de CPUs/GPUs y explotar toda su potencia de cómputo (CPUs y GPUs disponibles). En particular, nos centramos en aplicaciones desarrolladas con MPI+OpenMP+CUDA. Asimismo, explicamos los pasos a seguir para compilar estas aplicaciones híbridas. Resolvemos el problema de suma por reducción utilizando el esquema propuesto sobre un cluster de CPUs/GPUs heterogéneo, por esto la distribución de carga tiene en cuenta las capacidades de los recursos de cómputo disponibles. Comprobamos que es posible incrementar el rendimiento de la aplicación considerando todos los recursos de cómputo del cluster (CPUs y GPUs) respecto a utilizar sólo las GPUs.

Keywords: Programación híbrida, HPC, MPI, OpenMP, CUDA, Cluster, Multicore, GPU

1. Introducción

Hoy en día existen arquitecturas paralelas con gran poder de cómputo que combinan varios procesadores multi-core dentro de una misma máquina y múltiples GPUs. Para sacar provecho a esta arquitectura, las aplicaciones deben ser programadas a partir de combinar diferentes herramientas de programación, por ejemplo: MPI+CUDA, OpenMP+CUDA, Pthreads+CUDA [1,2,3,4]. En particular, en máquinas donde el número de cores supera el número de GPUs, la utilización de todos los recursos de cómputo disponibles (cores y GPUs) puede llevar a una mejora en el rendimiento con respecto a utilizar sólo un tipo de recurso.

Asimismo, varias de estas máquinas pueden conectarse a través de una red formando un cluster de CPUs/GPUs. En este caso, para explotar toda la potencia de cómputo provista por el cluster, la aplicación debe ser programada a partir de combinar herramientas de programación de memoria compartida y de paso de mensajes. Así surgen distintos patrones, como por ejemplo: MPI+OpenMP+CUDA y MPI+Pthreads+CUDA.

Aún más, las máquinas del cluster de CPUs/GPUs pueden diferir entre sí, es decir tanto en modelo y número de CPUs y GPUs, formando arquitecturas paralelas totalmente heterogéneas.

Estando en la era del "Big Data", una amplia gama de aplicaciones que procesan grandes volúmenes de datos para extraer información necesaria para la toma de decisiones en distintos ámbitos (salud, industria, finanzas, etc) pueden beneficiarse de las arquitecturas antes mencionadas para reducir su tiempo de cómputo.

Todo lo anterior supone un reto para el programador de aplicaciones, ya que no es trivial portar una aplicación heredada programada utilizando una única herramienta de programación paralela (MPI,Pthreads, OpenMP, CUDA) o combinaciones de dos de ellas, para que corra eficientemente sobre un cluster de CPUs/GPUs. Adicionalmente, la heterogeneidad del cluster de CPUs/GPUs añade un desafío adicional a la hora de distribuir la carga de trabajo.

En este trabajo describimos un esquema para estructurar código paralelo a ser ejecutado sobre un cluster de CPUs/GPUs, de modo de aprovechar toda la potencia de cómputo de esta arquitectura (CPUs y GPUs disponibles). En particular, nos centramos en aplicaciones desarrolladas con MPI+OpenMP+CUDA. Asimismo, explicamos los pasos a seguir para compilar estas aplicaciones híbridas. Resolvemos el problema de suma por reducción utilizando el esquema propuesto sobre un cluster de CPUs/GPUs heterogéneo, por esto la distribución de carga tiene en cuenta las capacidades de los recursos de cómputo disponibles. Comprobamos que es posible incrementar el rendimiento de la aplicación considerando todos los recursos de cómputo del cluster (CPUs y GPUs) respecto a utilizar sólo las GPUs.

El resto del paper está organizado de la siguiente manera. La Sección 2 presenta trabajos relacionados con la programación de GPUs, multi-GPUs y clusters de CPUs/GPUs. La Sección 3 introduce los modelos y herramientas de programación paralela. La Sección 4 introduce el esquema propuesto para estructurar código a ser ejecutado sobre un cluster de CPUs/GPUs. La Sección 5 muestra la evaluación experimental. Por último, la Sección 6 presenta las conclusiones y líneas de trabajo futuro.

2. Trabajos relacionados

En las últimas dos décadas, la evolución en las arquitecturas de cómputo trajo aparejada una revolución en la forma de programar las aplicaciones.

La integración de una GPU en una máquina ha impulsado el desarrollo de aplicaciones paralelas que explotan la potencia de ésta placa. En este caso, las herramientas de programación utilizadas son CUDA [4] u OpenCL [5], según el proveedor de la placa gráfica (NVIDIA/AMD). En particular, se pueden mencionar algoritmos de criptografía [6,7], análisis de imágenes médicas [8], simulación de dinámica de fluidos [9,10], etc, que alcanzan buena aceleración sobre una GPU.

Por otro lado, diversos autores han estudiado la paralelización de aplicaciones sobre clusters de CPUs/GPUs, debido a la atractiva relación costo/rendimiento de esta arquitectura. En general, el patrón utilizado al paralelizar se basa en lanzar un proceso MPI que controla a cada GPU disponible en el cluster. De este modo, se delega el control de la aplicación y la comunicación a MPI, mientras que se encarga a CUDA el cómputo intensivo de datos [9,11]. En particular, en [12] los autores paralelizan aplicaciones con MPI+CUDA y muestran cómo estructurar y compilar el código que combina estas dos herramientas. Para evaluar las aplicaciones, utilizan un cluster compuesto por dos máquinas heterogéneas (una de ellas integra 4 GPUs Tesla T10 y la otra 1 GPU Tesla C1060). La heterogeneidad subyacente en dicho cluster implica realizar la repartición de trabajo en base a la potencia de cómputo de las GPUs de cada máquina.

Otros autores han desarrollado aplicaciones que aprovechan la potencia de las múltiples GPUs integradas en una misma máquina. Esto da lugar a patrones híbridos (MPI+CUDA, OpenMP+CUDA, Pthreads+CUDA) que consisten en lanzar tantos procesos/threads como GPUs disponibles, encargando a cada uno la interacción con una GPU, la cual realiza el cómputo de datos [12,13,14].

Los modelos adoptados por los autores mencionados con anterioridad consisten en hacer trabajar a las GPUs disponibles, encargando la gestión de las GPUs a un número similar de cores de la arquitectura. En el caso particular que el número de cores disponibles en una máquina exceda al número de GPUs, estos modelos llevan a infrautilizar los recursos de cómputo, ya que ciertos cores de la máquina quedan ociosos.

En este sentido, en [15] los autores proponen un patrón híbrido MPI+OpenMP +CUDA (MOC) para aprovechar toda la potencia de cómputo de un cluster de CPUs/GPUs, y lo contrastan contra los patrones MPI puro y MPI+CUDA (MC). El patrón MOC se basa en lanzar un proceso MPI por cada GPU del cluster. Cada proceso a su vez genera una cantidad de threads adecuada (dada por mxnc/ng-1, donde m es el número de procesadores de la máquina, nc es la cantidad de cores de cada procesador y ng es la cantidad de GPUs de la máquina). El thread maestro del proceso se encargará de interactuar con la GPU, la cual realizará parte del cómputo sobre los datos, mientras que los demás threads computan el restante de datos. Para obtener buen rendimiento, la distribución de la carga de trabajo dentro de cada máquina la realizan teniendo en cuenta las capacidades de cómputo de la CPU y GPU. De los resultados experimentales obtenidos a partir de correr distintos tipos de aplicaciones sobre el cluster homogéneo TianHe-1A, en el cual cada nodo posee una única GPU, destacan que este patrón es efectivo especialmente para aplicaciones de cómputo intensivo.

Siguiendo una línea similar, el trabajo [16] introduce la paralelización del algoritmo de criptografía AES utilizando OpenMP+CUDA, sobre una máquina con múltiples cores y una única GPU. La estrategia propuesta por los autores consiste en lanzar varios hilos, uno de ellos se encarga de la interacción con la GPU y de invocar al kernel para computar AES sobre la GPU, mientras que el resto de los hilos computan AES sobre cores de la CPU. Para la repartición de trabajo tienen en cuenta la capacidad de la GPU y de la CPU. Comparan esta

estrategia híbrida (HPP y HPUP) contra aquellas que consisten en utilizar sólo la GPU (GPP y GPUP) y sólo los cores de la CPU (CPP y CPUP), programadas únicamente con CUDA y OpenMP respectivamente. La estrategia híbrida obtiene mejores resultados, a esta la sigue el algoritmo CUDA y por último el algoritmo OpenMP.

De lo anterior podemos destacar que, si bien en [15] proponen un patrón MPI+OpenMP+CUDA para aprovechar toda la potencia de un cluster de CPUs/GPUs, no indican la manera de estructurar el código híbrido ni tampoco muestran cómo realizar la compilación, la cual no es trivial. Además, los autores centran su experimentación sobre un cluster homogéneo, por lo tanto la distribución de carga es sencilla.

3. Modelos y Herramientas de Programación Paralela

Un modelo de programación paralela describe un sistema paralelo desde el punto de vista del programador. Una clasificación posible es según el mecanismo de interacción de los procesos:

- Modelo de memoria compartida: los procesos se comunican a través de escribir y leer la memoria compartida.
- Modelo de paso de mensajes: los procesos se comunican a través de enviar y recibir mensajes (ya que no tienen acceso a zonas de memoria compartida).

A continuación se resumen las características de las herramientas OpenMP, MPI y CUDA, hoy predominantes en el desarrollo de aplicaciones paralelas. Si bien existen otras herramientas de programación paralela, como por ejemplo Pthreads, Cilk, OpenCL, entre otras, no serán tratadas aquí, pero se discuten en detalle en [17].

3.1. OpenMP

OpenMP [2] es un estándar que define una API para la programación de aplicaciones sobre memoria compartida, disponible para los lenguajes C, C++ y Fortran. Provee un conjunto de directivas, rutinas y variables de entorno. A partir del conjunto de directivas se puede crear threads, realizar operaciones de sincronización y distribuir la carga de trabajo entre threads.

OpenMP impone una estructura jerárquica entre los threads, ya que sigue el modelo fork/join. En este modelo la ejecución del programa comienza como un único hilo (maestro), que ejecuta secuencialmente hasta encontrar una región paralela (directiva parallel). En ese momento, el maestro crea un conjunto de threads (fork). A partir de allí, cada thread ejecutará las sentencias encerradas en la región paralela en forma concurrente. Cuando todos los threads finalizaron la ejecución, se sincronizan y terminan (join), quedando sólo el thread maestro, el cual continúa la ejecución.

3.2. MPI

MPI (Message Passing Interface) [1] es un estándar que define una API para la comunicación por paso de mensajes. Existen diversas implementaciones (por ejemplo, OpenMPI y MPICH), que a su vez se adaptan a distintas arquitecturas, de este modo aprovechan las características físicas del sistema. Entre las funciones básicas que provee MPI se encuentran aquellas que posibilitan realizar comunicación punto a punto y comunicación colectiva.

3.3. CUDA

CUDA (Compute Unified Device Architecture) [4] es una extensión al lenguaje C que permite a los programadores escribir código a ser ejecutado en una GPU NVIDIA de forma sencilla.

Desde el punto de vista del programador CUDA, el sistema paralelo está formado por un host (CPU) y un device (GPU). Así, el programa CUDA tendrá fases a ejecutar en el host y fases a ejecutar en el device (codificadas a través de funciones llamadas kernels). En el device, el cómputo correspondiente a un kernel es realizado por un conjunto de hilos, que se organizan en bloques y que a su vez se agrupan en un grid. De esta manera, cuando el programador invoca a un kernel debe especificar la organización de los hilos (esto es, la cantidad de hilos por bloque y su organización, y la cantidad de bloques del grid y su organización). La ejecución del grid será tratada por una única GPU. En particular, cada Streamming Multiprocessor de la GPU será encargado de ejecutar uno o varios bloques de éste grid. Únicamente los hilos que conforman un bloque pueden cooperar, a través de la memoria compartida, y sincronizarse.

El modelo de memoria CUDA se organiza en una jerarquía y está compuesto por: la memoria global, que admite lectura/escritura por parte del host y del device; la memoria de constantes, que admite lectura/escritura por parte del host y sólo lectura por parte del device; la memoria compartida, que puede ser accedida por todos los hilos de un bloque; y por último los registros accesibles sólo por el hilo.

4. Esquema híbrido MPI+OpenMP+CUDA

En la actualidad, es posible encontrar arquitecturas paralelas integradas por dispositivos con características diferentes, por ejemplo, clusters de CPUs/GPUs compuestos por máquinas con múltiples cores y múltiples GPUs. A su vez, las CPUs y GPUs de las distintas máquinas pueden tener diferentes prestaciones. Con el objetivo de aprovechar al máximo la potencia de estas arquitecturas heterogéneas, las aplicaciones deben ser desarrolladas combinando distintas herramientas de programación paralela. El código de una aplicación con estas características en general es complejo y difícil de estructurar.

En esta sección describimos un esquema para estructurar código paralelo a ser ejecutado sobre un cluster de CPUs/GPUs, de modo de aprovechar todos los

cores y GPUs. En particular, nos centramos en aplicaciones desarrolladas con MPI+OpenMP+CUDA, en las cuales se pueden diferenciar tres tipos de código: el código MPI, donde se realiza la comunicación entre los procesos generados; el código OpenMP, encargado de crear los hilos, algunos de ellos realizarán el cómputo sobre los cores y otros gestionarán las GPUs; el código CUDA, encargado de crear los hilos que realizarán el cómputo sobre la GPU.

Para desarrollar una aplicación con estas características, la forma más sencilla es incluir todo el código dentro de un mismo archivo. Sin embargo, utilizaremos una implementación modular ya que facilita la lectura y el mantenimiento del código.

La estructura que proponemos implementa el código de cada herramienta en archivos separados, como explicamos a continuación.

El archivo principal *mpi_source.c* incluye únicamente código MPI. Al iniciar la ejecución de la aplicación, MPI genera tantos procesos como el usuario indique. En nuestro caso, generamos un proceso por máquina del cluster. Cada proceso ejecutará el código que se encuentra en dicho archivo, cuyo objetivo es el control de la aplicación, la comunicación entre máquinas y la distribución de datos. A su vez, cada proceso interactúa con el módulo que implementa el código OpenMP *omp_source.c*.

mpi_source.c

```
#include <mpi.h>
#include "omp_source.h"

static void mpi_function(int myRank){
  ...
  omp_function();
  ...
}

int main(int argc, char *argv[]){
  int myRank;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&myRank);
  mpi_function(myRank);
  MPI_Finalize();
  return(0);
}
```

omp_source.h

```
#ifndef _FUNCION_OMP_
#define _FUNCION_OMP_
  extern void omp_function();
#endif
```

omp_source.c

```
#include<omp.h>
#include "cuda_source.h"

void core_function(){
  ...
}

void omp_function(void){
  ...
  int tid = omp_get_thread_num();
#pragma omp parallel
{
  if(tid < cuda_deviceCount())
        cuda_function(tid);
  else
        core_function();
}
  ...
}
```

En el módulo OpenMP se crean tantos hilos como cores posea la máquina. Un número determinado de estos hilos se destina a gestionar las GPUs disponibles. En particular, se asocia un hilo por GPU y, por simplicidad, estos hilos son aquellos con menor identificador. De esta forma, el identificador del hilo coincide con el identificador de la GPU con la que se relaciona. El resto de los hilos se destinan al cómputo sobre los cores de la máquina. Los hilos que ges-

tionan las GPUs interactúan con el módulo que implementa el código CUDA *cuda_source.cu*.

El módulo CUDA implementa todo lo relacionado a la ejecución sobre una GPU: selecciona la GPU; aloca espacio en memoria de la GPU; realiza las copias host-to-device; determina la cantidad de hilos a utilizar y su organización (bloques y grid); invoca el kernel CUDA; por último, recupera los resultados (copia device-to-host) y libera la memoria alocada en la GPU.

cuda_source.h

```
#ifndef __FUNCION_CUDA__
#define __FUNCION_CUDA__
  extern void cuda_function(int gpuID);
  extern int cuda_deviceCount(void);
#endif
```

cuda_source.cu

```
#include<cuda.h>
#include<cuda_runtime.h>

__global__ void kernel_function(dtype* args){
  ...
}

extern "C" void cuda_function(int gpuID){
  ...
  cudaSetDevice(gpuID);
  cudaMalloc(...);
  cudaMemcpy(..., cudaMemcpyHostToDevice);
  dim3 dimGrid(block_size);
  dim3 dimBlock(grid_size);
  kernel_function<<<dimGrid, dimBlock>>>(args);
  cudaMemcpy(..., cudaMemcpyDeviceToHost);
  cudaFree(...);
  ...
}

extern "C" int cuda_deviceCount(void){
int nDevices;

  cudaGetDeviceCount(&nDevices);
  return nDevices;
}
```

Este esquema puede ser aplicado en la resolución de problemas regulares, en los cuales los datos a procesar se encuentran disponibles desde el inicio de la ejecución de la aplicación.

4.1. Compilación

Generalmente, el código implementado usando una herramienta particular se compila con un compilador específico. Por ejemplo: el código MPI se compila con *mpicc*; el código OpenMP se compila con *gcc*; y el código CUDA se compila con *nvcc*. Afortunadamente, *mpicc* y *nvcc* son "wrappers" de *gcc* y es posible utilizar alguno de ellos para compilar la aplicación híbrida.

Básicamente, existen dos formas de compilar una aplicación híbrida. La primera consiste en utilizar un sólo compilador. En nuestro ejemplo, la forma más simple de hacerlo es a través del compilador de CUDA (*nvcc*), como se indica a continuación.

```
$ nvcc -Xcompiler -fopenmp -I$MPI_PATH/include
 -L$MPI_PATH/lib -lmpi -o mpi_omp_cuda mpi_source.c
  omp_source.c cuda_source.cu
```

La segunda forma consiste en compilar cada módulo por separado a partir del compilador correspondiente, generando código objeto, y luego enlazar el código generado utilizando alguno de los compiladores mencionados. En nuestro ejemplo, el enlazado lo realizamos con el compilador de MPI (*mpicc*).

```
$ mpicc -c mpi_source.c -o mpi_source.o
$ gcc -fopenmp -c omp_source.c -o omp_source.o
$ nvcc -c cuda_source.cu -o cuda_source.o
$ mpicc mpi_source.o omp_source.o cuda_source.o
-L$CUDA_PATH/lib64 -lcudart -lgomp -o mpi_omp_cuda
```

5. Experimentos

Utilizamos el esquema propuesto en la Sección 4 para resolver el problema de suma por reducción sobre un cluster de CPUs/GPUs. Específicamente, el algoritmo realiza la suma de los elementos de un vector. Evaluamos esta implementación, a la que llamaremos SR_MOC, sobre un cluster heterogéneo compuesto por 4 máquinas, cuyas características se muestran en la Tabla 1.

	Modelo de CPU	#CPUs	Cores	Modelo de GPU	#GPUs	GPU cores
Máquina 1	Intel i5 2310 2.9Ghz	1	4	Nvidia Tesla C2075 1150Mhz	2	448
Máquina 2	Intel i5 2310 2.9Ghz	1	4	Nvidia GTX 560Ti 1685Mhz	2	384
Máquina 3	Intel G2020 2.90GHz	1	2	Nvidia GTX 480 1401Mhz	2	480
Máquina 4	Intel i7 3770K 3.50GHz	1	4	Nvidia GTX 960 1127Mhz	1	1024

Tabla 1: Características del cluster de CPUs/GPUs heterogéneo

	Porcentaje por máquina	Porcentaje a los cores	Porcentaje a las GPUs
Máquina 1	21 %	11 %	89 %
Máquina 2	16 %	14 %	86 %
Máquina 3	24 %	0 %	100 %
Máquina 4	39 %	11 %	89 %

Tabla 2: Porcentaje de carga de trabajo asignada a cada máquina del cluster.

En un escenario heterogéneo es necesario distribuir la carga de trabajo teniendo en cuenta las capacidades de cada unidad de procesamiento. Varios autores estudiaron este tema. Por las características de nuestra aplicación, nos basamos en las propuestas de distribución estática de [18,19]. Para distribuir la carga, obtienen la potencia de cada procesador con respecto al mejor procesador de la arquitectura (*pcr* o potencia de cómputo relativa). En función de la *pcr* determinan el porcentaje de la carga de trabajo que debe computar cada unidad de procesamiento.

A partir de ejecutar la aplicación en cada unidad de procesamiento (core/GPU) del cluster, obtuvimos la potencia de cómputo relativa (*pcr*) de cada una

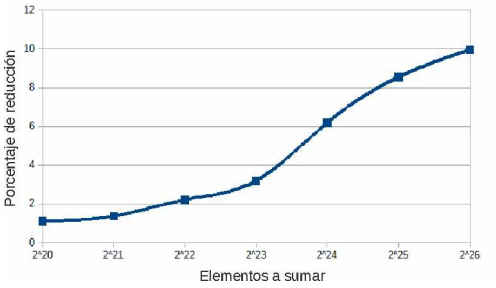

Figura 1: Reducción en el tiempo de ejecución SR_MOC vs SR_MC.

de las unidades. En función de las potencias calculadas, determinamos la *pcr* de cada máquina y, a partir de esto, el porcentaje de la carga de trabajo que recibirá cada una, el cual se muestra en la segunda columna de la Tabla 2.

Asimismo, la porción de la carga de trabajo recibida por una máquina debe distribuirse entre sus cores y GPUs de acuerdo a la potencia de cómputo relativa de cada uno de ellos dentro de esa máquina. De lo anterior, obtuvimos el porcentaje de la carga de trabajo que recibirá cada core/GPU de la máquina, los cuales se muestran en la tercera y cuarta columna de la Tabla 2.

Para ejemplificar esto último, podemos ver en la Tabla 1 que la Máquina 1 posee 4 cores y 2 GPUs: 2 cores estarán dedicados a interactuar con las GPUs y 2 cores estarán disponibles para realizar cómputo. De la porción de la carga de trabajo recibida por esta máquina, el 11 % será distribuido proporcionalmente entre los 2 cores que realizarán cómputo (5,5 % a cada uno) y el 89 % restante será procesado por las 2 GPUs (44,5 % a cada una). En el caso particular de la Máquina 3, los únicos 2 cores de la arquitectura estarán dedicados a gestionar las GPUs, por lo tanto las GPUs realizarán el 100 % del trabajo (50 % cada una).

Comparamos el algoritmo SR_MOC con un algoritmo MPI+CUDA que realiza la suma por reducción utilizando sólo las GPUs del cluster (sin tener en cuenta los cores), a la que llamaremos SR_MC. La Figura 1 muestra que el algoritmo SR_MOC reduce el tiempo de ejecución a medida que se incrementa el volumen de la carga de trabajo, con respecto al algoritmo SR_MC. Como se puede observar, el porcentaje de reducción es de hasta un 10 %.

6. Conclusiones y trabajo futuro

En este trabajo presentamos un esquema para estructurar código paralelo a ser ejecutado sobre un cluster de CPUs/GPUs, basado en MPI+OpenMP+CUDA. A diferencia de varios autores, que utilizan sólo las GPUs descartando los cores de la arquitectura, nuestra estructura permite desarrollar aplicaciones que explotan toda la potencia de cómputo del cluster (cores y GPUs).

Asimismo, explicamos los pasos a seguir para compilar estas aplicaciones híbridas, lo cual no es trivial.

Evaluamos el algoritmo de suma por reducción, desarrollado utilizando el esquema propuesto, sobre un cluster de CPUs/GPUs heterogéneo. Distribuimos la carga de trabajo de acuerdo a las capacidades de los recursos de cómputo disponibles. Concluimos que es posible incrementar el rendimiento de la aplicación hasta un 10 % si se consideran todos los recursos de cómputo del cluster (CPUs y GPUs) respecto a utilizar sólo las GPUs.

Como trabajo futuro, planeamos resolver otros problemas regulares utilizando el esquema propuesto y analizar su rendimiento sobre clusters hómogeneos y heterogeneos de CPUs/GPUs.

Referencias

1. Message Passing Interface Specification. http://www.mpi-forum.org
2. The OpenMP API specification for parallel programming. http://www.openmp.org
3. Posix Threads. http://standards.ieee.org/develop/wg/POSIX.html
4. Nvidia CUDA. http://www.nvidia.es/object/cuda-parallel-computing-es.html
5. OpenCL. https://www.khronos.org/opencl/
6. Manavski S.: CUDA Compatible GPU as an Efficient Hardware Accelerator for AES Cryptography. Proceedings of IEEE ICSPC 2007. 65–68 (2007)
7. Deguang L et al: Parallel AES algorithm for fast Data Encryption on GPU. Proceedings of ICCET 2010. 6, 1–6 (2010)
8. Eklunda A. et al: Medical image processing on the GPU – Past, present and future. Medical Image Analysis. 1073-–1094 (2013)
9. Fan Z. et al: GPU Cluster for High Performance Computing. Proceedings of the ACM - IEEE Supercomputing Conference 2004. (2004)
10. Harris M.: Fast Fluid Dynamics Simulation on the GPU. GPU Gems. Pearson (2007).
11. Kindratenko V. et al: GPU Clusters for High-Performance Computing. Proceedings of the IEEE International Conference on Cluster Computing and Workshops. 1–8 (2009)
12. Yang C. et al: Hybrid Parallel Programming on GPU Clusters. International Symposium on Parallel and Distributed Processing with Applications. 142–147 (2010)
13. Yang C. et al: Hybrid CUDA, OpenMP, and MPI parallel programming on multicore GPU clusters. Computer Physics Communications. 266-–269 (2011)
14. Noaje G. et al: MultiGPU computing using MPI or OpenMP. 347–354 (2010)
15. Fengshun L. et al: Performance evaluation of hybrid programming patterns for large CPU/GPU heterogeneous clusters. Computer Physics Communications. 1172—1181 (2012)
16. Xiongwei F. et al: A secure and efficient file protecting system based on SHA3 and parallel AES. Parallel Computing 52. 106-–132 (2016)
17. Diaz J. et al: A Survey of Parallel Programming Models and Tools in the Multi and Many-Core Era. IEEE Trans. Parallel Distrib. Syst. (2012)
18. AlJaroodi J. et al: Modeling Parallel Applications Performance on Heterogeneous System. Parallel and Distributed Processing Symposium, 2003. IEEEComputer Society. (2003)
19. Tinetti F. G., Tesis Doctoral: Cómputo Paralelo en Redes Locales de Computadoras. España: Universidad Autónoma de Barcelona. https://ddd.uab.cat/pub/tesis/2004/tdx-1027104-173002/fgt1de2.pdf